SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 9, 121–125 (1979

A Compact, Portable CRT-based Text Editor

CHRISTOPHER W. FRASER
Department of Computer Science, The University of Arizona, Tucson, Arizona 85721, U.S.A.

SUMMARY

CRT-based text editors offer a better user interface than most teletype-based text editors but are more complex and less portable. Building a screen editor as a front end to a line editor exploits existing code, yields a more compact, portable result, and permits one computer to edit another's files. This paper describes such an editor.

KEY WORDS Text editor CRT Front end Portable Ratfor

INTRODUCTION

Different text editors present different user interfaces but perform similar functions. Each prints, inserts, deletes and changes characters, moves a cursor about, searches for keys and performs file i/o. However, there are important differences in how different editors present text and locate the text to be changed.[1]

Line editors,[2] designed for slow hard-copy terminals, present text a line at a time and only on explicit request. Screen editors,[3] designed for fast CRT terminals, display text a screen at a time and update it after each editing operation so the user always views the current version of the text. This reduces the chance of error and spares the user the overhead of explicit requests to view lines.

The user of a line editor indicates target text with a line number or context. The editor may have a special name for the current line and support relative positioning with line number arithmetic, but the user must still name the target line. This often forces the user to follow a printed listing. The user of a screen editor specifies target text by pointing a cursor at it. Typed characters replace the indicated text immediately, and special characters delete or insert text immediately. Since the result is displayed immediately, editing errors are easily spotted and corrected.

Since they must do at least as much as a line editor, most screen editors consume more resources than line editors: more memory, more processor time, more development time. They stack a complex, terminal-specific screen handler atop an already file system-specific editor. The result is large and not easily transported. It is no wonder that screen editors are so rare even though users find them so useful. The rest of this paper illustrates techniques that avoid these problems and yield a compact, portable screen editor.

BUILDING ON A LINE EDITOR

Since screen and line editors perform similar functions, a screen editor can be built as a front end to a line editor. A front-end screen editor lets an existing line editor do the bulk of the editing and occupies itself with managing the screen and decoding commands. It translates screen editor commands into line editor commands, sends them to the underlying

0038–0644/79/0209–0121$01.00 *Received 20 February 1978*

line editor and displays the response at the appropriate screen position (Figure 1). Several advantages recommend this approach.

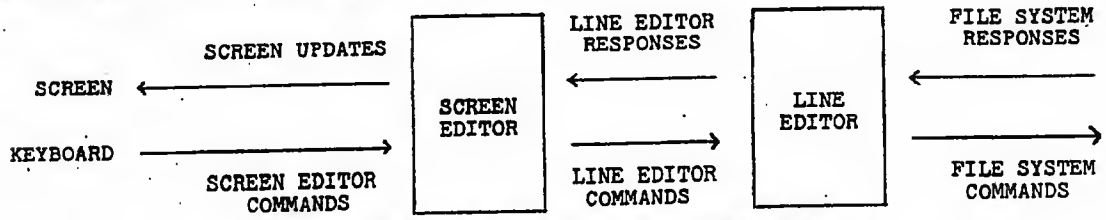

Figure 1. A screen editor front end to a line editor

Building on a line editor saves development time. A front-end screen editor need only decode commands and manage the screen. Existing line editor code performs the actual editing. The front end exploits an existing file system interface and so avoids many calls on an often-irritating operating system. It communicates with the file system in the line editor's command language using simple, well-documented, printable strings.

Building on a line editor improves portability. Few conventional, stand-alone screen editors are portable. This is easy to understand: they are, like other programs, somewhat machine-specific, but, unlike most other programs, they are also terminal-specific. A front-end screen editor is no less terminal-specific, though isolating the terminal handler in the compact front end may ease re-implementation. However, once this is transported, the complete screen editor may be moved by moving the underlying line editor—a straightforward task with the right editor[4]—or by exploiting a strategy made possible by the use of a front end. Most interactive systems support a line editor. Different systems offer different syntax though similar functions. So moving a front-end screen editor between machines may be accomplished by translating its twenty-odd line editor requests to the syntax of the line editor on the new machine; moving a conventional, stand-alone screen editor requires a new, non-portable operating system interface and, hence, experience with its file system. A front end may compensate for significant defects in the underlying line editor. For example, a screen editor's intraline editing facility (replacing overstruck characters) is easily built even if the underlying line editor has no such capability. A good screen editor is easily built on a simple line editor. However, some disparities between line and screen editor may be too great. For example, when scanning for a key, one line editor may terminate when it exhausts the file while another might wrap around and continue the search from the beginning of the file. Here, portability—that is, rapid re-implementation in a new environment—suffers unless the screen editor's search commands reflect those of the underlying line editor. A front-end screen editor should improve its line editor, not rewrite it.

Building on a line editor does, however, require that the front end and underlying line editor communicate. If interprocess communication is feasible, a front-end screen editor process may so communicate its commands to, and receive its results from, another line editor process. If a single-process version is required and if the line editor source code is available, the line editor may usually be transformed into a subroutine called by the front end. The unavailability of source code disqualifies many line editors. Luckily, portable line editors—with source code—are freely available.[4]

AN IMPLEMENTATION

The screen editor *s* was developed on a PDP10 as a front end to Kernighan and Plauger's portable line editor *edit*.[4] It now runs on a PDP11 as a front end to *edit* or to the *ed* text

editor[5] available under UNIX.[6] This history illustrates *s*'s portability. To the user, *s* resembles the Yale editor *E*.[3,7] Printable characters typed by the user overwrite the character under the screen cursor. Cursor control characters move the cursor about the screen. Control characters invoke commands that

—move the screen forward or backward in the file by any number of lines or pages or until a search key is found;
—insert or delete characters or lines;
—pick up text and put it elsewhere;
—change files.

s requires little of CRTs: it must be able to move the cursor up, down, right and left one position and to erase from the cursor to the end of line and end of screen. Most CRTs oblige. Terminal-dependent code is compact—roughly half a page—and isolated for easy modification.

edit is an elegant, yet conventional, line editor with commands to insert, delete and print lines; to perform string searches and substitutions; and to read and write files. It provides *s* with a portable operating system interface. *edit* is written in Ratfor[8] and comes with a preprocessor that translates Ratfor to a portable subset of FORTRAN. It needs only a few system-dependent routines: to read and write characters and lines; to create, open and close files; and—for best results—to delete files and to seek to the *n*th character of a disc file. Beyond this, *s* needs no more operating system interface than a routine which reads 'raw' characters from the terminal (i.e. gives no special interpretation to carriage return, backspace, etc.).

On the PDP11, *s* and *edit* processes communicate via a UNIX interprocess communications pipe[6] that makes *s* appear as a terminal to *edit*. No changes to *edit* were needed. On the PDP10, a multiprocess editor was not feasible, so *edit* was transformed into a subroutine called by *s*. This changed fewer than twenty lines of *edit*, but, of course, the source code had to be available. Portability, available source and excellent documentation recommend *edit*.

s uses resources efficiently. It was developed in two man-weeks. At 375 lines of Ratfor, *s* is quite short; in comparison, *edit* is about 1300 lines, and one of the shortest implementations of the stand-alone Yale editor[9] is about 1300 lines of IMP, a more compact language than Ratfor. For its improvements to *edit*'s interface, *s* requires about 50 per cent more memory and 5–50 per cent more processor time, depending on the editing session; for all practical purposes, it responds to requests as quickly as *edit*, and, because it requires less searching and fewer keystrokes, it may reduce connect time. Even though this implementation was designed to minimize development time rather than hardware costs, *s* competes well in its environment, and optimization does not seem to be worth the development effort. *s* has been moved—by its author—between terminals (VT05 to Ontel A4000, in two man-hours) and between machines (PDP10 to PDP11, in three man-days).

Its brevity makes *s* flexible enough for users to configure interfaces to suit themselves. For example, it would take only a few minutes to remove the modification commands and so produce a read-only version of *s* that would be safe for the uninitiated. It would require little more effort to add commands that can be rewritten as *edit* operations.

REMOTE EDITING

Programmers who use several different computers often wish to examine or even change files stored on a machine other than the one that they are presently logged onto. The situation at the University of Arizona is typical. A PDP11 offers inexpensive document

formatting, while a PDP10 supports more language processors; when preparing documents on the PDP11, a programmer may need to check the program—run, and hence stored, on the PDP10—being documented.

Fortunately, *s* and its offspring *edit* process need not run on the same machine. The PDP10 and PDP11 are linked with low-speed communications lines—the same 1200 baud hard-wired lines and 300 baud phone lines used to interface terminals to either machine—and so appear to one another as terminals. When asked to edit a remote file (flagged by a special name), the PDP11 *s* no longer creates a local *edit* process to accept and answer its commands, but instead may use such a communication line to log onto the PDP10 and create an *edit* process there. Looking like a PDP10 terminal running *edit*, *s* may send *edit* requests on, and receive *edit* responses over, this line. It no longer matters that the PDP10 operating system discourages multiprocess jobs: *s* on the PDP11 appears to the PDP10 as a terminal running *edit*—and only *edit*.

Some care must be taken in this; for example, escape characters from the remote file may be given special interpretations when they are received from a 'terminal'. However, once such details are overcome, the result is a contrivance not much slower than running *s* from the PDP10: since most of the data passing between *s* and *edit* is intended for the screen, it has to pass over low-speed lines whether it goes straight to a PDP10 terminal or via the PDP11 to a PDP11 terminal. Of course, this feature of *s* is not portable, since it relies on special file names, a particular hardware configuration, and the uniform mechanism UNIX provides for terminal i/o and interprocess communication.

Although little editing is now remote, economics may encourage changes. Terminals with limited local processing power (8-bit CPU, 4K bytes of memory) are now commercially available for as little as $600. Such a terminal could be dedicated to running *s*, and thereby present *edit* commands to the host processor while presenting a more pleasant interface to its user. To reduce communications over low-speed lines, such terminals might run an optimized *s*. For example, *s* now asks *edit* to search for keys; a terminal with local storage of the screen contents might first look there for the next occurrence of the key and so potentially save an *edit* request.

CONCLUSIONS

Many programs are written as stand-alone packages even when they might have been implemented as front ends to similar, though somewhat deficient, existing programs. The computational saving does not always justify the extra development effort. As hardware costs fall and programming costs rise, computational savings become less critical, and front ends become more attractive. Designers of operating systems can help by simplifying user multiprogramming.

Front ends reduce the volume of code and so reduce portability problems. They also offer alternatives on the continuum between full-bootstrap portability (where little target machine software is used) and half-bootstrap portability (where much target machine software is used). For example, *s* could be distributed without *edit* (requiring an existing line editor but little installation effort) or with *edit* (requiring no line editor but more installation effort).

The success of the front-end screen editor recommends other applications of this strategy. For example, the awkward interface presented by a dynamic debugger[10] might be buffered through a screen handler that displays and allows editing of a user-specified set of machine variables. Or an editor might be used as a front end to a document formatter

to allow editing of the formatted result instead of the intermediate file of text and formatter commands as is now common. Some such stand-alone systems already exist at isolated installations. They are more easily written, maintained and transported as front ends.

ACKNOWLEDGEMENTS

Work with Ned Irons on a related project uncovered the trouble spots in remote editing. Ralph Griswold and Dave Hanson suggested several improvements to earlier versions of this paper.

APPENDIX: *S* COMMANDS

This table outlines the *s* commands. All commands are initiated by a single character. For example, typing a printable character replaces the character at the screen cursor with the character typed and advances the cursor to the next character position. Typing the ASCII control character for '+pages' moves the screen's window on the file ahead one screen full.

character	*interpretation*
printable	replace character at cursor.
cursor move	move cursor without changing file or screen.
± pages	move forward or backward one screen full.
± lines	move forward or backward one line.
± search	move forward or backward until a key is found.
delete	delete one line; copy to special file.
insert	insert one blank line.
pick	copy one line to special file.
put	insert special file.
set file	edit new file.
exit	stop editing.

All commands accept an optional argument:

string	(e.g. *escape* foo *put* puts 'foo' after the cursor in the file and on the screen).
number	(e.g. *escape* 2 *+pages* moves forward 2 screen fulls).
cursor move	(e.g. *escape* movement *delete* deletes all characters and lines between the initial and final cursor positions).

REFERENCES

1. A. van Dam and D. E. Rice, 'On-line text editing: a survey', *Computing Surveys*, **3**, 93–114 (1976).
2. L. P. Deutsch and B. W. Lampson, 'An online editor', *CACM*, **10**, 793–799 (1967).
3. E. T. Irons and F. M. Djorup, 'A CRT editing system', *CACM*, **15**, 16–20 (1972).
4. B. W. Kernighan and P. J. Plauger, *Software Tools*, Addison-Wesley, Reading, Mass., 1976.
5. B. W. Kernighan, *A Tutorial Introduction to the ed Text Editor*, Technical Report, Bell Laboratories.
6. D. Ritchie and K. Thompson, 'The UNIX time-sharing system', *CACM*, **17**, 365–375 (1974).
7. P. Wiener, I. Singh, D. J. Mostow and E. T. Irons, *The Yale Editor E*, Research Report 19, Dept. of Computer Science, Yale University, 1973.
8. B. W. Kernighan, 'Ratfor—a preprocessor for a rational Fortran', *Software—Practice and Experience*, **5**, 395–406 (1975).
9. J. Meehan, *Documentation of the F Text Editor*, Dept. of Computer Science, University of California, Irvine, 1977.
10. *DDT—Dynamic Debugging Technique*, Digital Equipment Corporation, Publication DEC–10–UDDTA–A–D, 3rd ed., Maynard, Mass., 1974.